chapter-07-01

March 17, 2024

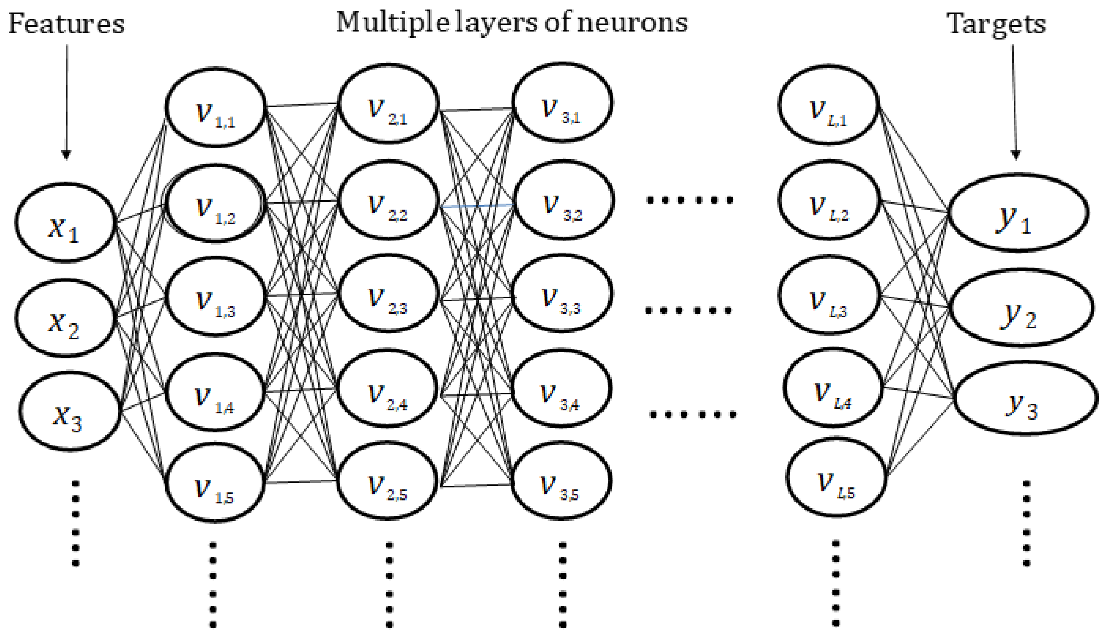
Run in Google Colab

1 Introduction to Deep Learning

1.1 What is a Neural Network

Neural networks, also known as artificial neural networks (ANNs) or simulated neural networks (SNNs), are a subset of machine learning and are at the heart of deep learning algorithms.

Artificial neural networks (ANNs) are comprised of a node layers, containing an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network.



Neural networks rely on **training data** to learn and improve their accuracy over time. However, once these learning algorithms are fine-tuned for accuracy, they are powerful tools in computer science and artificial intelligence, allowing us to classify and cluster data at a high velocity. Tasks in speech recognition or image recognition can take minutes versus hours when compared to the manual identification by human experts. One of the most well-known neural networks is Google's search algorithm.

1.1.1 Transformations

A Neural Network transforms its input data into meaningful outputs, a process that is **learned** from exposure to known examples of inputs and outputs. Therefore, the central problem in deep learning is to **meaningfully transform data**: in other words, **to learn useful representations of the input data at hand, representations that get us closer to the expected output**.

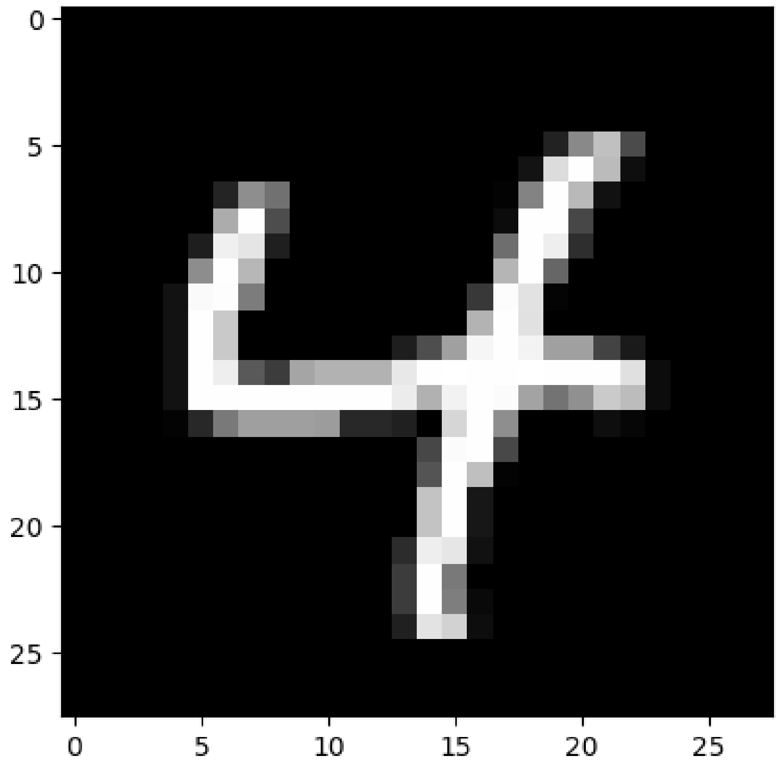
What is a representation? At its core, it is simply a different way to look at data, to represent or encode data.

```
In [1]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        data = pd.read_csv('./data/mnist_train.csv')
        data.head()
```

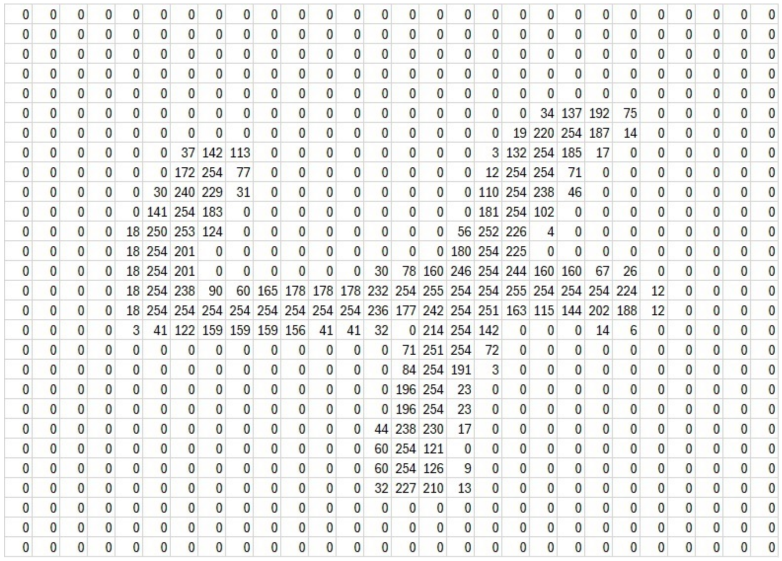
```
Out[1]:    label  1x1  1x2  1x3  1x4  1x5  1x6  1x7  1x8  1x9  …  28x19  28x20  \
        0      5    0    0    0    0    0    0    0    0    0  …      0      0
        1      0    0    0    0    0    0    0    0    0    0  …      0      0
        2      4    0    0    0    0    0    0    0    0    0  …      0      0
        3      1    0    0    0    0    0    0    0    0    0  …      0      0
        4      9    0    0    0    0    0    0    0    0    0  …      0      0

           28x21  28x22  28x23  28x24  28x25  28x26  28x27  28x28
        0      0      0      0      0      0      0      0      0
        1      0      0      0      0      0      0      0      0
        2      0      0      0      0      0      0      0      0
        3      0      0      0      0      0      0      0      0
        4      0      0      0      0      0      0      0      0

        [5 rows x 785 columns]
```

```
In [2]: x_train = np.array(data.iloc[:,1:])
        random_index =7178 #np.random.randint(0,40000)
        img = x_train[random_index].reshape(28,28)
        plt.imshow(img, cmap = "gray")
        print(random_index)
```

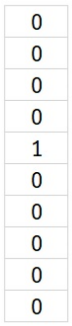
```
7178
```



```
In [3]: np.savetxt('./data/number.csv',img, delimiter=';')
```



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	34	137	192	75	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	19	220	254	187	14	0	0	0	0	0
0	0	0	0	0	0	37	142	113	0	0	0	0	0	0	0	0	3	132	254	185	17	0	0	0	0	0	0
0	0	0	0	0	0	172	254	77	0	0	0	0	0	0	0	0	12	254	254	71	0	0	0	0	0	0	0
0	0	0	0	0	30	240	229	31	0	0	0	0	0	0	0	0	110	254	238	46	0	0	0	0	0	0	0
0	0	0	0	0	141	254	183	0	0	0	0	0	0	0	0	0	181	254	102	0	0	0	0	0	0	0	0
0	0	0	0	18	250	253	124	0	0	0	0	0	0	0	0	56	252	226	4	0	0	0	0	0	0	0	0
0	0	0	0	18	254	201	0	0	0	0	0	0	0	0	0	180	254	225	0	0	0	0	0	0	0	0	0
0	0	0	0	18	254	201	0	0	0	0	0	0	30	78	160	246	254	244	160	160	67	26	0	0	0	0	0
0	0	0	0	18	254	238	90	60	165	178	178	178	232	254	255	254	254	255	254	254	254	224	12	0	0	0	0
0	0	0	0	18	254	254	254	254	254	254	254	254	236	177	242	254	251	163	115	144	202	188	12	0	0	0	0
0	0	0	0	3	41	122	159	159	159	156	41	41	32	0	214	254	142	0	0	0	14	6	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	71	251	254	72	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	84	254	191	3	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	196	254	23	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	196	254	23	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	44	238	230	17	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	60	254	121	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	60	254	126	9	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	32	227	210	13	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

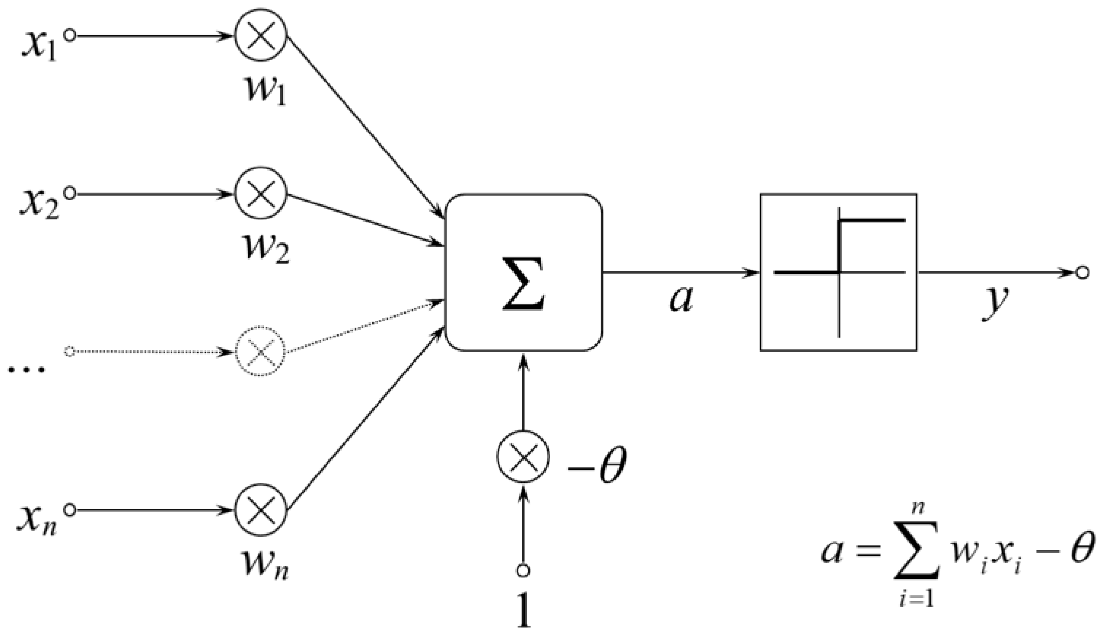


0
0
0
0
1
0
0
0
0
0

1.2 Mc Culloch and Pitts Artificial Neuron

The McCulloch and Pitts neuron is one of the oldest neural network. It has a single neuron and is the simplest form of a neural network. So it is very important to learn how it works because it is the most fundamental unit of a deep neural networks.

The artificial neuron receives one or more inputs and sums them to produce an output. Usually each input is separately weighted, and the sum is passed through a non-linear function known as an activation function or transfer function. The transfer functions usually have a sigmoid shape, but they may also take the form of other non-linear functions, piecewise linear functions, or step functions. They are also often monotonically increasing, continuous, differentiable and bounded.



1.3 Feedforward Neural Networks

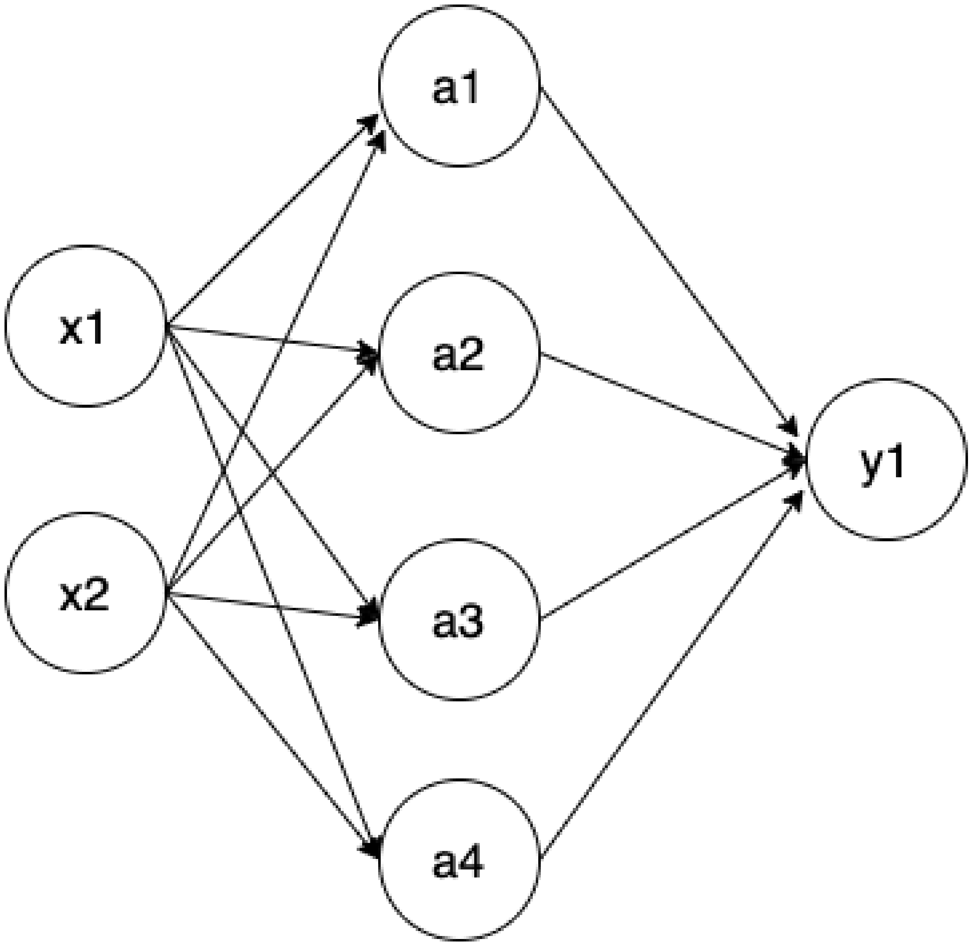
Feedforward neural networks, or multi-layer perceptrons (MLPs), are what we've primarily been focusing on within this notebook. They are comprised of an input layer, a hidden layer or layers, and an output layer. Data usually is fed into these models to train them, and they are the foundation for computer vision, natural language processing, and other neural networks.

The simplest kind of feedforward neural network is a single-layer network, which consists of a single layer of output nodes; the inputs are fed directly to the outputs via a series of weights. The sum of the products of the weights and the inputs is calculated in each node, and if the value is above some threshold the neuron fires and takes the activated value; otherwise it takes the deactivated value.

In the following we are going to implement a very simple neural network from scratch without any library. In my opinion this is very usefull because most people consider neural networks as a black-box and use libraries like Keras, TensorFlow and PyTorch which provide, among other things, automatic differentiation without a real understanding of how a neural network really works. Though it is not necessary to write your own code on how to compute gradients and backprop errors, having knowledge on it helps you in understanding a few concepts which can help you a lot in understanding how a neural networks works..

1.3.1 One Hidden Layer NN

We will build a shallow dense neural network with one hidden layer



Where in the graph above, we have a input vector $x = (x_1, x_2)$, containing 2 features and 4 hidden nodes a_1, a_2, a_3 and a_4, and only one value in output $y_1 \in [0, 1]$ (consider this a binary classification task with a prediction of probability)

In each hidden unit, take a_1 as example, a linear operation followed by an activation function, f, is performed. So given input $x = (x_1, x_2)$, inside node a_1, we have:

$$z_1 = w_{11}x_1 + w_{12}x_2 + b_1$$

$$a_1 = f(w_{11}x_1 + w_{12}x_2 + b_1) = f(z_1)$$

Here w_{11} denotes weight 1 of node 1, w_{12} denotes weight 2 of node 1. Same for node a_2, it would have:

$$z_2 = w_{21}x_1 + w_{22}x_2 + b_2$$

$$a_2 = f(w_{21}x_1 + w_{22}x_2 + b_2) = f(z_2)$$

And same for a_3 and a_4 and so on ...

We can also write in a more compact form

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]} \tag{1}$$

Note that superscript $[i]$ denotes the *ith* layer. Let's assume that the first activation function is the tanh and the output activation function is the *sigmoid*. So the result of the hidden layer is:

$$A^{[1]} = \tanh Z^{[1]}$$

This result is applied to the output node which will perform another linear operation with a different set of weights, $W^{[2]}$:

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + B^{[2]}$$

and the final output will be the result of the application of the output node activation function (the sigmoid) to this value:

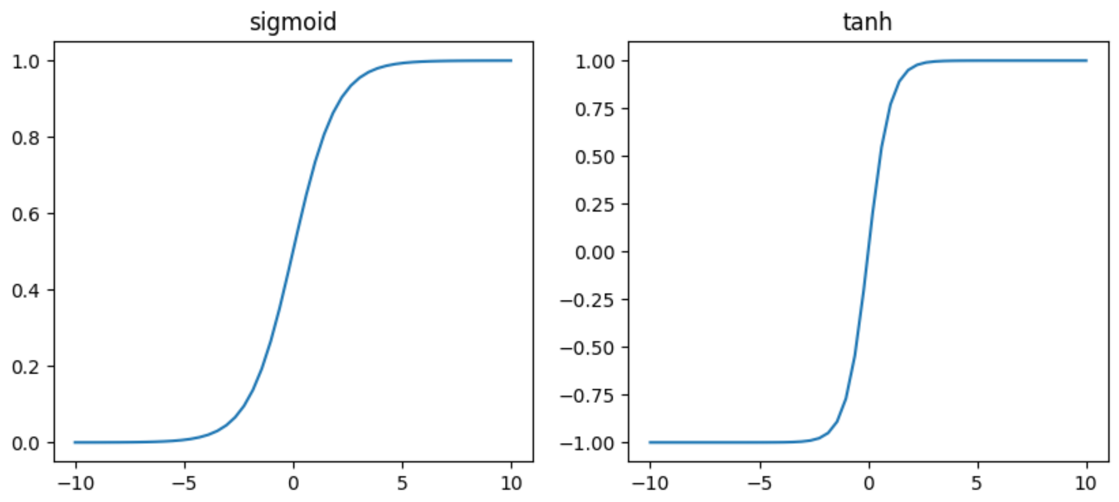
$$\hat{y} = \sigma(Z^{[2]})$$

For the dimension of each matrix, we have:

- $ W^{[1]}$ in the case above would have dimension 4×2, with each *ith* row is the weight of node i
- $B^{[1]}$ has dimension 4×1
- $Z^{[1]}$ and $A^{[1]}$ both have dimention 4×1
- $W^{[2]}$ has dimension 1×4
- consequently, $Z^{[2]}$ and $A^{[2]}$ would have dimensition 1×1, which is a single value

Function tanh and *sigmoid* looks as below.

```
In [4]: %matplotlib inline

        import numpy as np
        import matplotlib.pyplot as plt

In [5]: def tanh(x):
            return np.tanh(x)

        def sigmoid(x):
            return 1/(1 + np.exp(-x))

In [6]: plt.figure(figsize=[10, 4])
        x = np.linspace(-10, 10)

        plt.subplot(1, 2, 1)
        plt.plot(x, sigmoid(x))
        plt.title('sigmoid')

        plt.subplot(1, 2, 2)
        plt.plot(x, tanh(x))
        plt.title('tanh')

Out[6]: Text(0.5, 1.0, 'tanh')
```



Notice that the only difference of these functions is the scale of y

For the derivatives of the activation functions the following rules apply (see notebook chapter-07-01 appendix for details) :

Derivative of Hyperbolic Tangent

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} \Rightarrow \frac{d}{dx} \tanh x = 1 - (\tanh x)^2 \tag{2}$$

Derivative of Sigmoid Function

$$\sigma(x) = \left[\frac{1}{1+e^{-x}}\right] \Rightarrow \frac{d}{dx}\sigma(x) = \sigma(x) \cdot (1 - \sigma(x)) \tag{3}$$

1.3.2 The Loss Function

Remember that Linear regression uses Least Squared Error as loss function that gives a convex graph and then we can complete the optimization by finding its vertex as global minimum. However, it's not an option for logistic regression anymore. Since the hypothesis is changed, Least Squared Error will result in a non-convex graph with local minimums by calculating with sigmoid function applied on raw model output.

Furthermore using logistic regression, means that we are focusing on binary classification, we have class 0 and class 1. To compare with the target, we want to constrain predictions to some values between 0 and 1. That's why Sigmoid Function is applied on the raw model output and provides the ability to predict with probability. So will follow a different path.

Intuitively, we want to assign more punishment when predicting 1 while the actual is 0 and when predict 0 while the actual is 1. The loss function of logistic regression is doing this exactly which is called Logistic Loss. If y = 1, when prediction = 1, the cost must be = 0, otherwise, when prediction = 0, the learning algorithm is punished by a very large cost. Similarly, if y = 0, predicting 0 has no punishment but predicting 1 has a large value of cost. In formula we have

$$L(y, \hat{y}) = \begin{cases} -\log \hat{y} & \text{when } y = 1 \\ -\log(1-\hat{y}) & \text{when } y = 0 \end{cases} \tag{4}$$

Another advantage of this loss function is that although we are looking at it by y = 1 and y = 0 separately, it can be written as one single formula which brings convenience for calculation:

$$L(y, \hat{y}) = -[y \log \hat{y} + (1-y) \log(1-\hat{y})]$$

1.3.3 Formula of Batch Training

The above shows the formula of a single input vector, however in actual training processes, a batch is trained instead of 1 at a time. The change applied in the formula is trivial, we just need to replace the single vector x with a matrix X with size $n \times m$, where n is number of features and m is the the batch size, samples are stacked column wise, and the following result matrix are applied likewise. We have the same formulas as before ...

$$\begin{cases} Z^{[1]} = W^{[1]}X + b^{[1]} \\ A^{[1]} = \tanh Z^{[1]} \end{cases} \tag{5}$$

and

$$\begin{cases} Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \\ A^{[2]} = \sigma(Z^{[2]}) = \hat{y} \end{cases} \tag{6}$$

... but for the dimension of each matrix taken in this example now we have:

- X has dimension $2 \times m$, as here there are 2 features and m is the batch size
- $W^{[1]}$ in the case above would have dimension 4×2, with each ith row is the weight of node i
- $b^{[1]}$ has dimension 4×1
- $Z^{[1]}$ and $A^{[1]}$ both have dimension $4 \times m$
- $W^{[2]}$ has dimension 1×4
- consequently, $Z^{[2]}$ and $A^{[2]}$ would have dimension $1 \times m$

The loss function is the same as logistic regression, but for batch training, we'll take the average loss for all training samples.

$$J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m}\sum_{i}^{m} L(y^{(i)}, \hat{y}^{(i)})$$

This is all for the forward propagation. To activate our neurons to learn, we need to get derivative of weight parameters and update them use gradient descent.

But now it is enough for us to implement the forward propagation first.

1.3.4 Generate Sample Dataset

Scikit-learn includes various random sample generators that can be used to build artificial datasets of controlled size and complexity. Here we generate a simple **binary classification task** with 5000 data points and `n_features` features for later model validation.

```
In [7]: n_features = 20
        n_hidden   = 10
```

```
In [8]: from sklearn import datasets
        # Define the parameters for generating synthetic data using the make_classification function.
        # The function is invoked with specific arguments:
        # - n_samples    = 5000 : It generates a dataset with 5000 samples.
        # - random_state = 123  : It sets the random seed to ensure reproducibility.
        # - X is a NumPy array containing the feature data.
        # - y is a NumPy array containing the corresponding target labels.
        X, y = datasets.make_classification(n_samples=5000, n_features=n_features, n_informative=n_featu
        # Split the dataset into training and testing subsets: X_train and y_train contain 4000 samples
        # which are used for training the machine learning model. X_test and y_test contain 1000 sample
        # which are used for testing the model's
        X_train, X_test = X[:4000], X[4000:]
        y_train, y_test = y[:4000], y[4000:]
        print('train shape', X_train.shape)
        print('test shape', X_test.shape)
```

```
train shape (4000, 20)
test shape (1000, 20)
```

```
In [9]: print(X[0])
```

```
[-0.28726786  4.48392645 -0.19663023  0.52875293 -1.44603544  3.33013291
 -0.49828395 -1.38991547  3.86964059  0.0259337   1.41931003 -0.26677859
  3.21847861 -0.67232203 -2.7307319  -0.85791875  0.0631985   2.39609383
 -0.39872069 -3.49423158]
```

```
In [10]: print(y[0:20])
```

```
[0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 1 0 0 0 1]
```

1.3.5 Weights Initialization

Our neural network has 1 hidden layer and 2 layers in total(hidden layer + output layer), so there are 4 weight matrices to initialize ($W^{[1]}, b^{[1]}$ and $W^{[2]}, b^{[2]}$). Notice that the weights are initialized relatively small so that the gradients would be higher thus learning faster in the beginning phase.

```
In [11]: '''
         This code defines a Python function init_weights that initializes the weights and biases for a
         network with a specified architecture. The function takes three arguments:
         - n_input: An integer representing the number of input features.
         - n_hidden: An integer representing the number of neurons (units) in the hidden layer.
         - n_output: An integer representing the number of neurons in the output layer.

         The function returns a dictionary params containing the initialized weights and biases for the
         '''
         def init_weights(n_input, n_hidden, n_output):
             # Initializes an empty dictionary called params where we will store the weight and bias pa
             params = {}
             #  Initializes the weight matrix W1 for the first layer. It uses NumPy's np.random.randn f
             # to generate random numbers from a standard normal distribution (mean 0, standard deviati
             # and multiplies them by 0.01 to scale the values. The shape of W1 is (n_hidden, n_input),
             # which corresponds to the number of hidden units and input features.
             params['W1'] = np.random.randn(n_hidden, n_input) * 0.01
             #  Initializes the bias vector b1 for the first layer. It sets all the bias values to zero
             # The shape of b1 is (n_hidden, 1).
             params['b1'] = np.zeros((n_hidden, 1))
             # Initializes the weight matrix W2 for the second (output) layer in a similar way as W1.
             # The shape of W2 is (n_output, n_hidden).
             params['W2'] = np.random.randn(n_output, n_hidden) * 0.01
             #  Initializes the bias vector b2 for the second layer, also setting all the bias values t
             # The shape of b2 is (n_output, 1).
             params['b2'] = np.zeros((n_output, 1))

             return params
```

```
In [12]: params = init_weights(n_features, n_hidden, 1)

         print('W1 shape', params['W1'].shape)
         print('b1 shape', params['b1'].shape)
         print('W2 shape', params['W2'].shape)
         print('b2 shape', params['b2'].shape)
```

```
W1 shape (10, 20)
b1 shape (10, 1)
W2 shape (1, 10)
b2 shape (1, 1)
```

1.3.6 Forward Propagation

Let's implement the forward process following equations (5) ∼ (8).

```
In [13]: '''
         This code defines a Python function named forward that performs the forward propagation step o
         Forward propagation computes the activations of each layer given the input data and the networ
         The function takes two arguments:

         - X      : A NumPy array representing the input data. The shape is expected to be (n_features 
                    where n_features is the number of input features, and m_samples is the number of da

         - params : A dictionary containing the network's parameters, including the weight matrices and
                    each layer. These parameters are typically initialized using the init_weights funct
         '''
         def forward(X, params):
             # Unpacks the parameters from the params dictionary
             W1, b1, W2, b2 = params['W1'], params['b1'], params['W2'], params['b2']
             # Initializes A0 as the input data
             A0 = X
             # Initializes a dictionary cache to store intermediate values during forward propagation
             cache = {}
             #
             # Computes the linear transformation and activation for the first layer (hidden layer):
             #
             # Compute the linear transformation Z1 by taking the dot product of the weight matrix W1 a
             # input A0, and adding the bias vector b1:
             Z1 = np.dot(W1, A0) + b1
             # Compute the activation A1 of the first layer by applying the hyperbolic tangent (tanh)
             # activation function to Z1:
             A1 = tanh(Z1)
             #
             # Computes the linear transformation and activation for the second layer (output layer):
             #
             # Compute the linear transformation Z2 by taking the dot product of the weight matrix W2 a
             # activation A1 from the first layer, and adding the bias vector b2:
             Z2 = np.dot(W2, A1) + b2
             # Compute the activation A2 of the second layer by applying the sigmoid activation functio
             A2 = sigmoid(Z2)
             # Stores the computed values in the cache dictionary for later use, typically in the backp
             cache['Z1'] = Z1
             cache['A1'] = A1
```

```
        cache['Z2'] = Z2
        cache['A2'] = A2
        return  cache
```

```
In [14]: # get 100 samples
         inp = X[:100].T

         cache = forward(inp, params)

         print('Z1 shape', cache['Z1'].shape)
         print('A1 shape', cache['A1'].shape)
         print('Z2 shape', cache['Z2'].shape)
         print('A2 shape', cache['A2'].shape)
```

```
Z1 shape (10, 100)
A1 shape (10, 100)
Z2 shape (1, 100)
A2 shape (1, 100)
```

1.3.7 Loss Function

Following equation (9), let's calculate the loss of each batch.

```
In [15]: '''
         The loss function calculates the binary cross-entropy loss, which is commonly used as a loss f
         binary classification problems. It quantifies the dissimilarity between the true values and th
         values, with lower values indicating a better match between predictions and ground truth.

         The function takes two arguments:

         - Y    : A NumPy array representing the true values. It is expected to be a vector (1-dimensio
         - Y_hat: A NumPy array representing the predicted values. It is also expected to be a vector w
         '''
         def loss(Y, Y_hat):
             # This assertion checks that Y is a row vector (1 row, multiple columns). It ensures that
             # values are organized as a row vector.
             assert Y.shape[0] == 1
             # This assertion checks that the shape of Y matches the shape of Y_hat, which means they s
             # the same number of elements. This is essential to compare true and predicted values.
             assert Y.shape == Y_hat.shape
             # calculates the number of samples, denoted as m, by getting the number of columns in the
             m = Y.shape[1]
             # computes the binary cross-entropy loss. We calculate the loss for each individual sample
             # vectors Y and Y_hat. Then we sum the log likelihood of the true class (if Y is 1) and th
             # likelihood of the complementary class (if Y is 0) for each sample. Finally, we compute t
             # loss by taking the negative sum of the s values and dividing by the number of samples m.
             # This is the average loss over all samples:
             s = Y * np.log(Y_hat) + (1 - Y) * np.log(1 - Y_hat)
             loss = -np.sum(s) / m
             return loss
```

```
In [16]: Y = np.array([np.random.choice([0, 1]) for i in range(10)]).reshape(1, -1)
         Y_hat = np.random.uniform(0, 1, 10).reshape(1, -1)
```

```
        l = loss(Y, Y_hat)
        print(f'loss {l}')
```

```
loss 1.383534612058882
```

1.3.8 Delta Rule

Now it comes to the so called ***delta rule*** which is the key to our weights update. With *delta rule* we compute the gradient of the loss function with respect to the weights of the network for a single input–output example.

Given a generic actual value y, we want to minimize the loss L, and the technic we are going to apply here is gradient descent, basically what we need to do is to apply derivative to our variables and move them slightly down to the optimum. Here we have 2 variables, W and b, and for this example, the update formula of them would be:

$$W_{new} = W_{old} - \frac{\partial L}{\partial W} \Rightarrow \Delta W = -\frac{\partial L}{\partial W} \tag{7}$$

$$b_{new} = b_{old} - \frac{\partial L}{\partial b} \Rightarrow \Delta b = -\frac{\partial L}{\partial b} \tag{8}$$

The delta rule algorithm works by computing the gradient of the loss function with respect to each weight. In order to get the derivative of our targets, chain rules would be applied:

$$\Delta W^{[1]} = -\frac{\partial L}{\partial W^{[1]}} \tag{9}$$

$$\Delta W^{[2]} = -\frac{\partial L}{\partial W^{[2]}} \tag{10}$$

$$\frac{\partial L}{\partial W^{[i]}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial Z^{[2]}} \frac{\partial Z^{[2]}}{\partial W^{[i]}} \tag{11}$$

where $i = 1, 2$

Given the loss function L we defined above, we can easily compute the first partial derivative with respect to $\hat{y}$:

$$L(y, \hat{y}) = -[y \log \hat{y} + (1-y) \log(1-\hat{y})] \Rightarrow \frac{\partial L}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}$$

Remeber that in our notation we have:

$$\hat{y} = \sigma\left(Z^{[2]}\right) = A^{[2]}$$

So we can write for the two first partial derivatives of our chain:

$$\frac{\partial L}{\partial \hat{y}} = \frac{A^{[2]} - y}{A^{[2]}\left(1 - A^{[2]}\right)} \tag{12}$$

$$\frac{\partial \hat{y}}{\partial Z^{[2]}} = \frac{\partial \sigma\left(Z^{[2]}\right)}{\partial Z^{[2]}} = \sigma\left(Z^{[2]}\right) \cdot \left[1 - \sigma\left(Z^{[2]}\right)\right] = \hat{y}\left(1-\hat{y}\right) = A^{[2]}\left(1 - A^{[2]}\right) \tag{13}$$

putting it all together

$$\frac{\partial L}{\partial W^{[i]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial W^{[i]}} \tag{14}$$

$$= \frac{A^{[2]} - y}{A^{[2]}\left(1 - A^{[2]}\right)} \cdot A^{[2]}\left(1 - A^{[2]}\right) \cdot \frac{\partial Z^{[2]}}{\partial W^{[i]}} \tag{15}$$

$$= \left(A^{[2]} - y\right) \cdot \frac{\partial Z^{[2]}}{\partial W^{[i]}} \tag{16}$$

where $i = 1, 2$

Now we have only to compute the partial derivatives of the pre-activation function $Z^{[2]}$ with respect to $W^{[1]}$ and $W^{[2]}$.

Remember that

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + B^{[2]} = W^{[2]} \cdot \tanh\left(W^{[1]} \cdot X + B^{[1]}\right) + B^{[2]}$$

where

$$A^{[1]} = \tanh\left(W^{[1]} \cdot X + B^{[1]}\right) = \tanh\left(Z^{[1]}\right)$$

We have

$$\frac{\partial Z^{[2]}}{\partial W^{[2]}} = A^{[1]} \tag{17}$$

$$\frac{\partial Z^{[2]}}{\partial W^{[1]}} = W^{[2]} \cdot \frac{\partial \tanh\left(Z^{[1]}\right)}{\partial Z^{[1]}} \cdot \frac{\partial Z^{[1]}}{\partial W^{[1]}} \tag{18}$$

$$\begin{cases} \dfrac{\partial \tanh\left(Z^{[1]}\right)}{\partial Z^{[1]}} = 1 - \left[\tanh\left(Z^{[1]}\right)\right]^2 = 1 - \left(A^{[1]}\right)^2 \\ \dfrac{\partial Z^{[1]}}{\partial W^{[1]}} = X \end{cases} \tag{19}$$

$$\frac{\partial Z^{[2]}}{\partial W^{[1]}} = W^{[2]} \cdot X \cdot \left[1 - \left(A^{[1]}\right)^2\right]$$

Now we have all the elements to calculate the derivative of the loss function with respect to the weights

$$\frac{\partial L}{\partial W^{[1]}} = \left(A^{[2]} - y\right) \cdot \frac{\partial Z^{[2]}}{\partial W^{[1]}} = \left(A^{[2]} - y\right) \cdot W^{[2]} \cdot X \cdot \left[1 - \left(A^{[1]}\right)^2\right] \tag{20}$$

$$\frac{\partial L}{\partial W^{[2]}} = \left(A^{[2]} - y\right) \cdot \frac{\partial Z^{[2]}}{\partial W^{[2]}} = \left(A^{[2]} - y\right) \cdot A^{[1]} \tag{21}$$

In the python code we put also:

$$\Delta^{[2]} = A^{[2]} - y \tag{22}$$

$$\Delta^{[1]} = \Delta^{[2]} \cdot W^{[2]} \cdot (1 - A^{[1]^2}) \tag{23}$$

So we finally have

$$\Delta W^{[2]} = -\frac{\partial L}{\partial W^{[2]}} = -\Delta^{[2]} \cdot A^{[1]} \tag{24}$$

$$\Delta W^{[1]} = -\frac{\partial L}{\partial W^{[1]}} = -\Delta^{[1]} \cdot X \tag{25}$$

For completeness we also report the formulas for updating the bias vectors

$$\Delta b^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True) \tag{26}$$

$$\Delta b^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True) \tag{27}$$

In summary:

- Error is calculated between the expected outputs and the outputs forward propagated from the network.
- These errors are then propagated backward through the network from the output layer to the hidden layer, assigning a penalty for the error and updating weights as they go.

Let's break down the shape of each element, given number of each layer equals `(n_x, n_h, n_y)` and batch size equals `m`:

- $A^{[2]}$, Y and $dZ^{[2]}$ has shape `(n_y, m)`
- Because $A^{[1]}$ has shape `(n_h, m)`, $dW^{[2]}$ would have shape `(n_y, n_h)`
- $db^{[2]}$ has shape `(n_y, 1)`
- Because $dZ^{[2]}$ has shape `(n_y, m)`, $W^{[2]}$ has shape`(n_y, n_h)`, $dZ^{[1]}$ would have shape `(n_h, m)`
- In equation (5), X has shape `(n_x, m)`, so $dW^{[1]}$ has shape `(n_h, n_x)`
- $db^{[1]}$ has shape `(n_h, 1)`

Once we understand the formula, implementation should come with ease.

```
In [17]: def backward(params, cache, X, Y):
             """
             [From coursera deep-learning course]
             params: we initiate above with W1, b1, W2, b2
             cache: the intermediate caculation we saved with Z1, A1, Z2, A2
             X: shape of (n_x, m)
             Y: shape (n_y, m)
             """
             # Calculate the number of training examples m from the shape of the input data X:
             m = X.shape[1]
             # Extract the necessary parameters and intermediate values from the provided params and ca
             W1 = params['W1']
             W2 = params['W2']
             A1 = cache['A1']
             A2 = cache['A2']
             #
             # Compute the gradients for the parameters in the backward pass of backpropagation
             #
             # DL2 represents the derivative of the loss with respect to the activations of the output
             # layer (A2) and is calculated as the difference between A2 and Y:
             DL2 = A2 - Y
             # dW2 represents the gradient of the loss with respect to the weights of the second layer
             dW2 = (1 / m) * np.dot(DL2, A1.T)
             # db2 represents the gradient of the loss with respect to the biases of the second layer
             db2 = (1 / m) * np.sum(DL2, axis=1, keepdims=True)
             # DL1 represents the derivative of the loss with respect to the activations of the first
             # hidden layer (A1) and is computed using the chain rule. It involves the element-wise pro
             # of the transpose of W2 and the derivative of the activation function applied to A1:
             DL1 = np.multiply(np.dot(W2.T, DL2), 1 - np.power(A1, 2))
             # dW1 represents the gradient of the loss with respect to the weights of the first layer
             dW1 = (1 / m) * np.dot(DL1, X.T)
             # db1 represents the gradient of the loss with respect to the biases of the first layer
             db1 = (1 / m) * np.sum(DL1, axis=1, keepdims=True)

             grads = {"dW1": dW1,
                      "db1": db1,
                      "dW2": dW2,
                      "db2": db2}

             return grads
```

1.3.9 Batch Loader

Now let's ensemble everything into a class.

```
In [43]: class ShallowNN:
             def __init__(self, n_input, n_hidden, n_output):
                 self.n_input = n_input
                 self.n_hidden = n_hidden
                 self.n_output = n_output
                 self.params = {}
                 self.cache = {}
                 self.grads = {}

             def compute_loss(self, Y, Y_hat):
                 """
                 Y: vector of true value
                 Y_hat: vector of predicted value
                 """
                 assert Y.shape[0] == 1
                 assert Y.shape == Y_hat.shape
                 m = Y.shape[1]
                 s = Y * np.log(Y_hat) + (1 - Y) * np.log(1 - Y_hat)
                 loss = -np.sum(s) / m
                 return loss


             def init_weights(self):
                 self.params['W1'] = np.random.randn(self.n_hidden, self.n_input) * 0.01
                 self.params['b1'] = np.zeros((self.n_hidden, 1))
                 self.params['W2'] = np.random.randn(self.n_output, self.n_hidden) * 0.01
                 self.params['b2'] = np.zeros((self.n_output, 1))


             def forward(self, X):
                 """
                 X: need to have shape (n_features x m_samples)
                 """
                 W1, b1, W2, b2 = self.params['W1'], self.params['b1'], self.params['W2'], self.params[
                 A0 = X

                 Z1 = np.dot(W1, A0) + b1
                 A1 = tanh(Z1)
                 Z2 = np.dot(W2, A1) + b2
                 A2 = sigmoid(Z2)

                 self.cache['Z1'] = Z1
                 self.cache['A1'] = A1
                 self.cache['Z2'] = Z2
                 self.cache['A2'] = A2
```

```
def backward(self, X, Y):
    """
    [From coursera deep-learning course]
    params: we initiate above with W1, b1, W2, b2
    cache: the intermediate caculation we saved with Z1, A1, Z2, A2
    X: shape of (n_x, m)
    Y: shape (n_y, m)
    """

    m = X.shape[1]

    W1 = self.params['W1']
    W2 = self.params['W2']
    A1 = self.cache['A1']
    A2 = self.cache['A2']

    dZ2 = A2 - Y
    dW2 = (1 / m) * np.dot(dZ2, A1.T)
    db2 = (1 / m) * np.sum(dZ2, axis=1, keepdims=True)
    dZ1 = np.multiply(np.dot(W2.T, dZ2), 1 - np.power(A1, 2))
    dW1 = (1 / m) * np.dot(dZ1, X.T)
    db1 = (1 / m) * np.sum(dZ1, axis=1, keepdims=True)

    self.grads = {"dW1": dW1,
                  "db1": db1,
                  "dW2": dW2,
                  "db2": db2}


def get_batch_indices(self, X_train, batch_size):
    n = X_train.shape[0]
    indices = [range(i, i+batch_size) for i in range(0, n, batch_size)]
    return indices


def update_weights(self, lr):
    W1, b1, W2, b2 = self.params['W1'], self.params['b1'], self.params['W2'], self.params[
    dW1, db1, dW2, db2 = self.grads['dW1'], self.grads['db1'], self.grads['dW2'], self.gra
    self.params['W1'] -= dW1
    self.params['W2'] -= dW2
    self.params['b1'] -= db1
    self.params['b2'] -= db2


def fit(self, X_train, y_train, batch_size=32, n_iterations=100, lr=0.01):
    self.init_weights()

    indices = self.get_batch_indices(X_train, batch_size)
    for i in range(n_iterations):
        for ind in indices:
            X = X_train[ind, :].T
            Y = y_train[ind].reshape(1, batch_size)
```

```
                    self.forward(X)
                    self.backward(X, Y)
                    self.update_weights(lr)

                if i % 10 == 0:
                    Y_hat = self.cache['A2']
                    loss = self.compute_loss(Y, Y_hat)
                    print(f'iteration {i}: loss {loss}')


        def predict(self, X):
            W1, b1, W2, b2 = self.params['W1'], self.params['b1'], self.params['W2'], self.params[
            A0 = X

            Z1 = np.dot(W1, A0) + b1
            A1 = tanh(Z1)
            Z2 = np.dot(W2, A1) + b2
            A2 = sigmoid(Z2)

            return A2


    def accuracy(Y, Y_pred):
        """
        Y: vector of true value
        Y_pred: vector of predicted value
        """
        def _to_binary(x):
            return 1 if x > .5 else 0

        assert Y.shape[0] == 1
        assert Y.shape    == Y_pred.shape
        Y_pred = np.vectorize(_to_binary)(Y_pred)
        acc = float(np.dot(Y, Y_pred.T) + np.dot(1 - Y, 1 - Y_pred.T))/Y.size
        return acc
```

```
In [44]: model = ShallowNN(n_features, n_hidden, 1)
```

```
In [45]: model.fit(X_train, y_train, batch_size=100, n_iterations=500, lr=0.01)
```

```
iteration 0: loss 0.4281490830301316
iteration 10: loss 0.2650594799795105
iteration 20: loss 0.21221242844348837
iteration 30: loss 0.15905846229006518
iteration 40: loss 0.16790814386195776
iteration 50: loss 0.17437091535228144
iteration 60: loss 0.15809404888924816
iteration 70: loss 0.15838953416799326
iteration 80: loss 0.16173770228719916
iteration 90: loss 0.14751710162070725
iteration 100: loss 0.14834518865845428
iteration 110: loss 0.15421073574008853
iteration 120: loss 0.1542521885135119
iteration 130: loss 0.1411115381422983
iteration 140: loss 0.13839347433126867
iteration 150: loss 0.14012830844316546
iteration 160: loss 0.13232055542953133
iteration 170: loss 0.13428028978615641
iteration 180: loss 0.13205225850301347
iteration 190: loss 0.1329222510500098
iteration 200: loss 0.13052867692961123
iteration 210: loss 0.12961504273256572
iteration 220: loss 0.12676821496376126
iteration 230: loss 0.12904850785932515
iteration 240: loss 0.13170617232970291
iteration 250: loss 0.1324521246447653
iteration 260: loss 0.1294640285895979
iteration 270: loss 0.13039579704212526
iteration 280: loss 0.12608456759671538
iteration 290: loss 0.1274973550505577
iteration 300: loss 0.1268232404211177
iteration 310: loss 0.12777797767906296
iteration 320: loss 0.12620355466155814
iteration 330: loss 0.1260031748405518
iteration 340: loss 0.1263258092777216
iteration 350: loss 0.12699031593934418
iteration 360: loss 0.12698040356520982
iteration 370: loss 0.12632839107908028
iteration 380: loss 0.1253339235160142
iteration 390: loss 0.12666550456975487
iteration 400: loss 0.1252128228815929
iteration 410: loss 0.12376001033305271
iteration 420: loss 0.1238831209783091
iteration 430: loss 0.12359285133089566
iteration 440: loss 0.12270028060989768
iteration 450: loss 0.12258547937533333
iteration 460: loss 0.1230561564526091
iteration 470: loss 0.12276097707227195
iteration 480: loss 0.12234590035365031
iteration 490: loss 0.12210005201414478
```

```
In [46]: y_preds = model.predict(X_test.T)

         acc = accuracy(y_test.reshape(1, -1), y_preds)
         print(f'accuracy: {acc*100}%')
```

```
accuracy: 93.8%
```

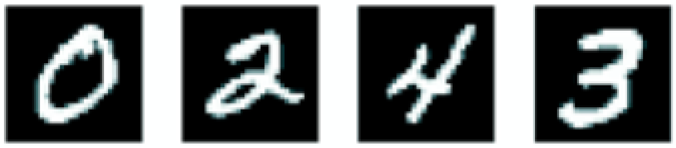
```
C:\Users\User\AppData\Local\Temp\ipykernel_9340\3223901223.py:134: DeprecationWarning: Conversion of an
  acc = float(np.dot(Y, Y_pred.T) + np.dot(1 - Y, 1 - Y_pred.T))/Y.size
```

1.4 Keras: the Python Deep Learning API

In this chapter we will present the code samples found in Chapter 2, Section 1 of **Deep Learning with Python** by François Chollet the creator of ***Keras***.

Keras is a high-level Deep Learning API that allows you to easily build, train, evaluate and execute all sorts of neural networks. Its documentation (or specification) is available at https://keras.io. It was developed by ***François Chollet*** as part of a research project and released as an open source project in March 2015. It quickly gained popularity owing to its ease-of-use, flexibility and beautiful design. To perform the heavy computations required by neural networks, keras-team relies on a computation backend. At the present, you can choose from three popular open source deep learning libraries: TensorFlow, Microsoft Cognitive Toolkit (CNTK) or Theano.

The problem we are trying to solve here is to classify grayscale images of handwritten digits (28 pixels by 28 pixels), into their 10 categories (0 to 9). The dataset we will use is the MNIST dataset, a classic dataset in the machine learning community, which has been around for almost as long as the field itself and has been very intensively studied. It's a set of 60,000 training images, plus 10,000 test images, assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s. You can think of "solving" MNIST as the "Hello World" of deep learning – it's what you do to verify that your algorithms are working as expected. As you become a machine learning practitioner, you will see MNIST come up over and over again, in scientific papers, blog posts, and so on.



```
In [47]: import keras
```

The MNIST dataset comes pre-loaded in Keras, in the form of a set of four Numpy arrays:

```
In [48]: from keras.datasets import mnist

         (train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

`train_images` and `train_labels` form the "training set", the data that the model will learn from. The model will then be tested on the "test set", `test_images` and `test_labels`. Our images are encoded as Numpy arrays, and the labels are simply an array of digits, ranging from 0 to 9. There is a one-to-one correspondence between the images and the labels.

Let's have a look at the training data:

```
In [49]: train_images.shape

Out[49]: (60000, 28, 28)

In [50]: len(train_labels)

Out[50]: 60000

In [51]: train_labels

Out[51]: array([5, 0, 4, …, 5, 6, 8], dtype=uint8)
```

Let's have a look at the test data:

```
In [52]: print(test_images.shape)
         print(test_labels.shape)

(10000, 28, 28)
(10000,)
```

```
In [53]: print(len(test_labels))
         print(test_labels[0])

10000
7
```

```
In [54]: test_labels

Out[54]: array([7, 2, 1, …, 4, 5, 6], dtype=uint8)
```

Our workflow will be as follow: first we will present our neural network with the training data, `train_images` and `train_labels`. The network will then learn to associate images and labels. Finally, we will ask the network to produce predictions for `test_images`, and we will verify if these predictions match the labels from `test_labels`.

Let's build our network. The code below defines a simple neural network model with two dense layers. The first layer has 512 neurons with ReLU activation, and the second layer has 10 neurons with softmax activation, which makes it suitable for multi-class classification tasks. This model can be used for tasks such as image classification, where input images are 28x28 pixels and there are 10 possible classes to predict.

```
In [55]: from keras import models
         from keras import layers

         # This line creates a sequential model, which is a linear stack of layers. In a sequential mod
         # can add layers one after another, and the data flows through the layers from input to output
         network = models.Sequential()
         # This line adds a dense layer to the model. Here's what each argument means:
         # - 512                    : This is the number of neurons (units) in the layer. The layer has
         #                            and this is the output dimension of the layer.
         # - activation='relu'      : This specifies that the ReLU (Rectified Linear Unit) activation f
         #                            for this layer. ReLU is a popular activation function that introd
         #                            into the network.
         # - input_shape=(28 * 28,) : This specifies the shape of the input data that the model expects
         #                            it's a flattened 28x28 image, so it has 784 input features.
         # This dense layer takes the 784 input features, processes them through 512 neurons with ReLU
         # and produces a 512-dimensional output.
         network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
         # This line adds a second dense layer to the model with 10 neurons. This layer uses the softma
         # The softmax activation is often used in the output layer of a classification model. It conve
         # raw output values into a probability distribution over multiple classes. In this case, it's
         # classification, where the network is expected to predict one of 10 possible classes.
         network.add(layers.Dense(10, activation='softmax'))
```

Let's go through this code line by line:

- The first line creates a Sequential model. This is the simplest kind of Keras model, for neural networks that are just composed of a single stack of layers, connected sequentially. This is called the sequential API.
- Next, we build the first layer and add it to the model. It is ***Dense*** hidden layer with 512 neurons. It will use the ReLU activation function. Each Dense layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs. It also manages a vector of bias terms (one per neuron). When it receives some input data, it computes

$$\phi\left(Z^{[1]} = W^{[1]} \cdot X + B^{[1]}\right), \quad \phi(z) = ReLU(z)$$

- Finally, we add a Dense output layer with 10 neurons (one per class). Using a 10-way "softmax" layer means that it will return an array of 10 probability scores (summing to 1). Each score will be the probability that the current digit image belongs to one of our 10 digit classes.

  NOTE - The ReLU Activation Function The Rectified Linear Unit (ReLU) activation function is a simple but widely used nonlinear activation function in artificial neural networks. It is defined as:

  $$f(x) = max(0, x)$$

  In other words, ReLU returns 0 for any negative input, and for positive input, it returns the same value. This function introduces sparsity in the network, as it zeros out negative values, which can help in reducing the likelihood of vanishing gradients during training. ReLU is computationally efficient and has been found to be effective in many deep learning architectures.

  NOTE - The softmax Activation Function The softmax activation function is commonly used in multi-class classification problems. It takes a vector of arbitrary real-valued scores as input and converts them into probabilities that sum up to 1. Mathematically, the softmax function is defined as follows:

  $$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$$

  where x_i represents the input score for class i, N is the total number of classes, and e is Euler's number (approximately 2.71828). The softmax function essentially ***transforms the input scores into a probability distribution over multiple classes***. This is achieved by exponentiating each score to ensure non-negativity and then normalizing the resulting values to sum up to 1. The higher the input score for a particular class relative to the other classes, the higher the probability assigned to that class by the softmax function. Softmax is typically used as the output activation function in the last layer of a neural network for multi-class classification tasks. It helps in making the network output interpretable as probabilities, which facilitates decision-making based on the most likely class prediction. Additionally, softmax enables the use of cross-entropy loss, which is commonly used as the objective function for training classification models.

The model's summary() method displays all the model's layers, including each layer's name (which is automatically generated unless you set it when creating the layer), its output shape (None means the batch size can be anything), and its number of parameters. The summary ends with the total number of parameters, including trainable and non-trainable parameters. Here we only have trainable parameters:

```
In [56]: network.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 512)               401920

 dense_1 (Dense)             (None, 10)                5130

=================================================================
Total params: 407050 (1.55 MB)
Trainable params: 407050 (1.55 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Note that Dense layers often have a lot of parameters. For example, **the first hidden layer has 784 × 512 connection weights, plus 512 bias terms, which adds up to 401920 parameters!** This gives the model quite a lot of flexibility to fit the training data, but it also means that the model runs the risk of overfitting, especially when you do not have a lot of training data. We will come back to this later.

To make our network ready for training, we need to pick three more things, as part of "compilation" step:

- A loss function: the is how the network will be able to measure how good a job it is doing on its training data, and thus how it will be able to steer itself in the right direction.
- An optimizer: this is the mechanism through which the network will update itself based on the data it sees and its loss function.
- Metrics to monitor during training and testing. Here we will only care about accuracy (the fraction of the images that were correctly classified).

see here for a description of rmsprop and here for crossentropy, note that for a binary classification, where the number of classes M equals 2, cross-entropy is exactly the loss function of the previous example.

```
In [57]: network.compile(optimizer='rmsprop',
                         loss='categorical_crossentropy',
                         metrics=['accuracy'])
```

Before training, we will preprocess our data by reshaping it into the shape that the network expects, and scaling it so that all values are in the `[0, 1]` interval. Previously, our training images for instance were stored in an array of shape `(60000, 28, 28)` of type `uint8` with values in the `[0, 255]` interval. We transform it into a `float32` array of shape `(60000, 28 * 28)` with values between 0 and 1.

```
In [58]: train_images = train_images.reshape((60000, 28 * 28))
         train_images = train_images.astype('float32') / 255

         test_images = test_images.reshape((10000, 28 * 28))
         test_images = test_images.astype('float32') / 255
```

We also need to categorically encode the labels:

```
In [59]: print(test_labels[0])
```

```
7
```

```
In [60]: from keras.utils import to_categorical

         train_labels = to_categorical(train_labels)
         test_labels = to_categorical(test_labels)
```

```
In [61]: print(test_labels[0])
```

```
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
```

What is an epoch?

An epoch is a term used in machine learning and indicates the number of passes of the entire training dataset the machine learning algorithm has completed. Datasets are usually grouped into batches (especially when the amount of data is very large). Some people use the term iteration loosely and refer to putting one batch through the model as an iteration.

If the batch size is the whole training dataset then the number of epochs is the number of iterations. For practical reasons, this is usually not the case. Many models are created with more than one epoch. The general relation where dataset size is d, number of epochs is e, number of iterations is i, and batch size is b would be de = ib.

Determining how many epochs a model should run to train is based on many parameters related to both the data itself and the goal of the model, and while there have been efforts to turn this process into an algorithm, often a deep understanding of the data itself is indispensable.

We are now ready to train our network, which in Keras is done via a call to the `fit` method of the network: we "fit" the model to its training data.

```
In [62]: network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

```
Epoch 1/5
469/469 [==============================] - 5s 9ms/step - loss: 0.2639 - accuracy: 0.9250
Epoch 2/5
469/469 [==============================] - 4s 8ms/step - loss: 0.1080 - accuracy: 0.9676
Epoch 3/5
469/469 [==============================] - 4s 9ms/step - loss: 0.0709 - accuracy: 0.9788
Epoch 4/5
469/469 [==============================] - 4s 9ms/step - loss: 0.0524 - accuracy: 0.9840
Epoch 5/5
469/469 [==============================] - 4s 8ms/step - loss: 0.0384 - accuracy: 0.9885
```

```
Out[62]: <keras.src.callbacks.History at 0x1baf8fb1360>
```

Two quantities are being displayed during training: the "loss" of the network over the training data, and the accuracy of the network over the training data.

We quickly reach an accuracy of 0.989 (i.e. 98.9%) on the training data. Now let's check that our model performs well on the test set too:

```
In [63]: test_loss, test_acc = network.evaluate(test_images, test_labels)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.0707 - accuracy: 0.9773
```

```
In [64]: print('test_acc:', test_acc)
```

```
test_acc: 0.9772999882698059
```

Our test set accuracy turns out to be 97.8% – that's quite a bit lower than the training set accuracy. This gap between training accuracy and test accuracy is an example of "overfitting", the fact that machine learning models tend to perform worse on new data than on their training data.

```
In [65]: X = np.concatenate((train_images, test_images))
         Y = np.concatenate((train_labels, test_labels))
         print(X.shape, Y.shape)
```

```
(70000, 784) (70000, 10)
```

```
In [66]: history = network.fit(X, Y, validation_split=0.33, epochs=5, verbose=1)
```

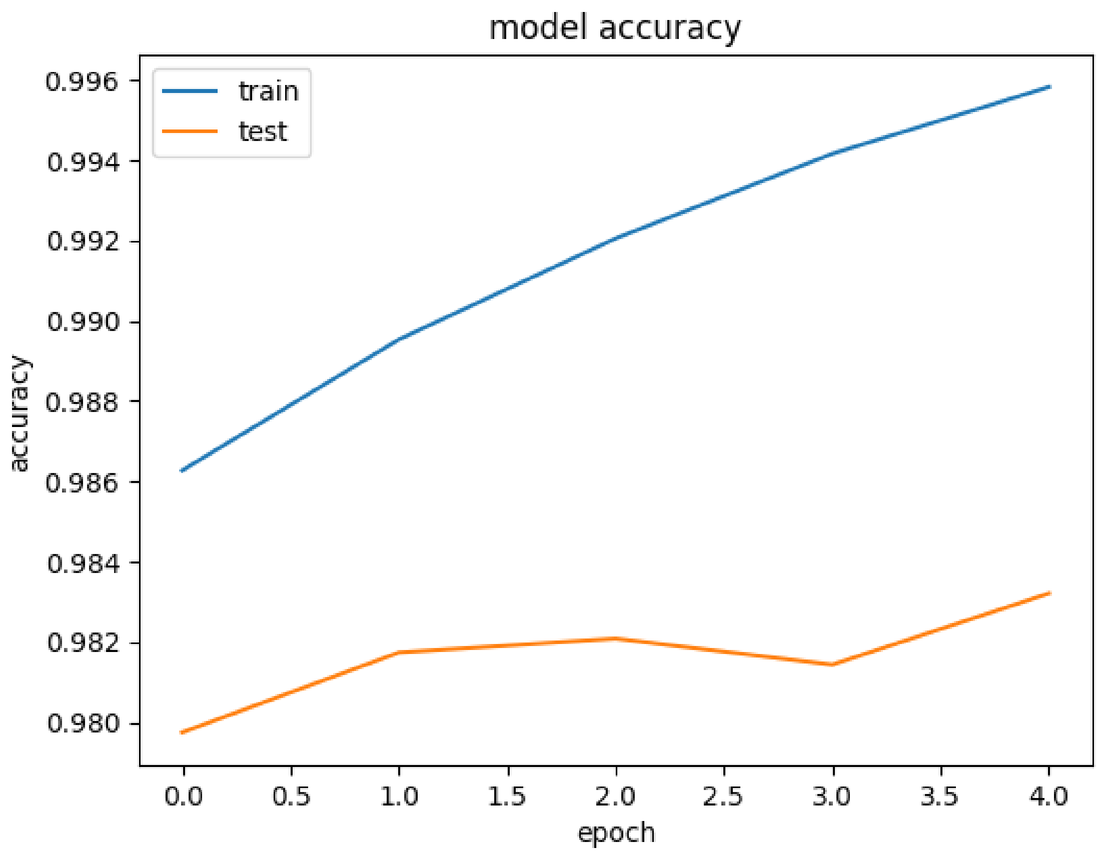
```
Epoch 1/5
1466/1466 [==============================] - 11s 8ms/step - loss: 0.0436 - accuracy: 0.9863 - val_loss:
Epoch 2/5
1466/1466 [==============================] - 12s 8ms/step - loss: 0.0339 - accuracy: 0.9895 - val_loss:
Epoch 3/5
```

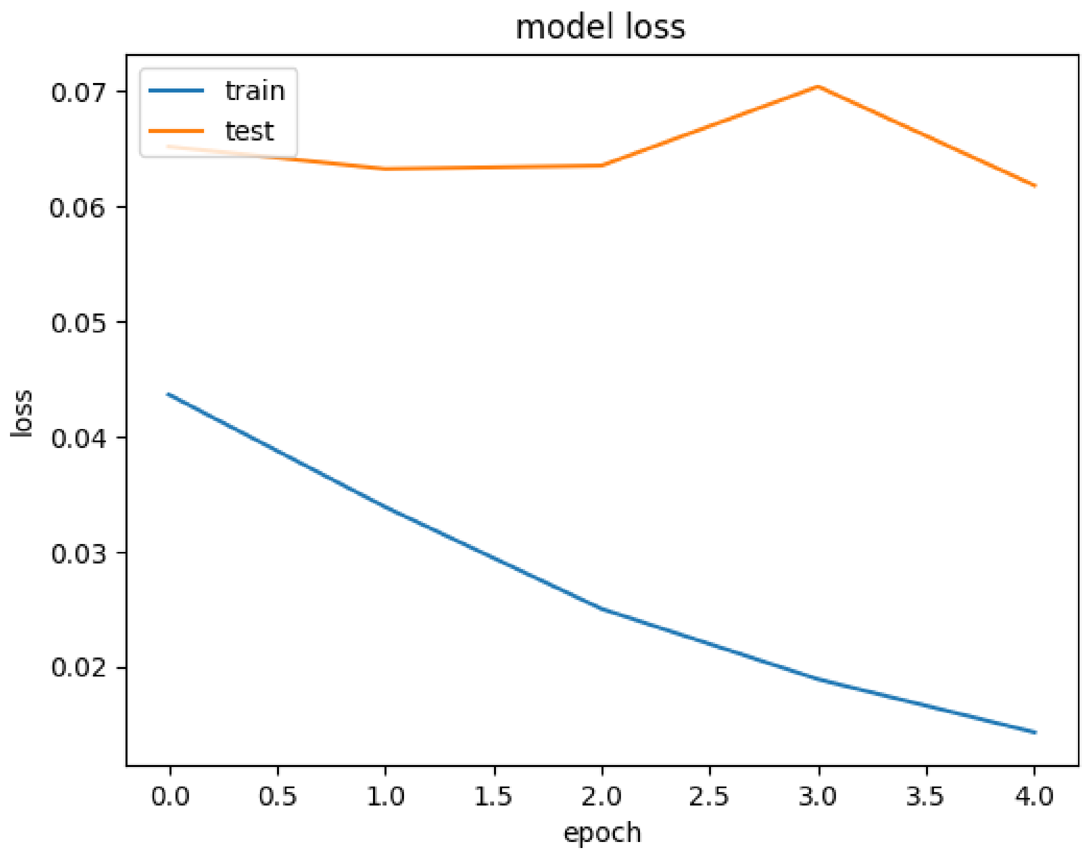
```
1466/1466 [==============================] - 11s 8ms/step - loss: 0.0250 - accuracy: 0.9920 - val_loss:
Epoch 4/5
1466/1466 [==============================] - 11s 8ms/step - loss: 0.0189 - accuracy: 0.9942 - val_loss:
Epoch 5/5
1466/1466 [==============================] - 13s 9ms/step - loss: 0.0143 - accuracy: 0.9958 - val_loss:
```

```
In [67]: # list all data in history
         print(history.history.keys())
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
In [68]: # summarize history for accuracy
         plt.plot(history.history['accuracy'])
         plt.plot(history.history['val_accuracy'])
         plt.title('model accuracy')
         plt.ylabel('accuracy')
         plt.xlabel('epoch')
         plt.legend(['train', 'test'], loc='upper left')
         plt.show()
         # summarize history for loss
         plt.plot(history.history['loss'])
         plt.plot(history.history['val_loss'])
         plt.title('model loss')
         plt.ylabel('loss')
         plt.xlabel('epoch')
         plt.legend(['train', 'test'], loc='upper left')
         plt.show()
```


model accuracy
train
test
accuracy
epoch
0.980
0.982
0.984
0.986
0.988
0.990
0.992
0.994
0.996
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0



This concludes our very first example: you just saw how we could build and a train a neural network to classify handwritten digits, in less than 20 lines of Python code.

1.5 Other types of Neural Networks

Convolutional neural networks (CNNs) are similar to feedforward networks, but they're usually utilized for image recognition, pattern recognition, and/or computer vision. These networks harness principles from linear algebra, particularly matrix multiplication, to identify patterns within an image.

Recurrent neural networks (RNNs) are identified by their feedback loops. These learning algorithms are primarily leveraged when using time-series data to make predictions about future outcomes, such as stock market predictions or sales forecasting.

1.6 Appendix - Calculation of Activation Functions Derivatives

1.6.1 Derivative of the Hyperbolic Tangent

$$\frac{d}{dx}\tanh x = \frac{d}{dx}\left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right) = \frac{\left(e^x + e^{-x}\right)\frac{d}{dx}\left(e^x - e^{-x}\right) - \left(e^x + e^{-x}\right)\frac{d}{dx}\left(e^x - e^{-x}\right)}{\left(e^x + e^{-x}\right)^2}$$

$$\begin{aligned}
\frac{d}{dx}(\tanh x) &= \frac{(e^x+e^{-x})(e^x+e^{-x}) - (e^x+e^{-x})(e^x+e^{-x})}{(e^x+e^{-x})^2} \\
\Rightarrow \frac{d}{dx}(\tanh x) &= \frac{(e^x+e^{-x})^2 - (e^x+e^{-x})^2}{(e^x+e^{-x})^2} \\
\Rightarrow \frac{d}{dx}(\tanh x) &= \frac{(e^{2x}+e^{-2x}+2e^xe^{-x}) - (e^{2x}+e^{-2x} - 2e^xe^{-x})}{(e^x+e^{-x})^2} \\
\Rightarrow \frac{d}{dx}(\tanh x) &= \frac{(e^{2x}+e^{-2x}+2) - (e^{2x}+e^{-2x} - 2)}{(e^x+e^{-x})^2} \\
\Rightarrow \frac{d}{dx}(\tanh x) &= \frac{e^{2x}+e^{-2x}+2 - e^{2x} - e^{-2x}+2}{(e^x+e^{-x})^2} \\
\Rightarrow \frac{d}{dx}(\tanh x) &= \frac{4}{(e^x+e^{-x})^2} \\
\Rightarrow \frac{d}{dx}(\tanh x) &= 1-\left(\frac{e^x-e^{-x}}{e^x+e^{-x}}\right)^2 = 1-(\tanh x)^2
\end{aligned} \tag{28}$$

1.6.2 Derivative of the Sigmoid

$$\begin{aligned}
\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right] \\
&= \frac{d}{dx}(1+\mathrm{e}^{-x})^{-1} \\
&= -(1+e^{-x})^{-2}(-e^{-x}) \\
&= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{(1+e^{-x})-1}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \cdot \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right) \\
&= \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) \\
&= \sigma(x)\cdot(1-\sigma(x))
\end{aligned}$$

1.7 References and Credits

Chollet F., “*Deep Learning with Python*” Manning (2018)

Jeremy Z., “*Build a Shallow Neural Network*” click here for the original post.